SSA-AC: Static Significance Analysis for Approximate Computing

SARA AYMAN METWALLI and YUKO HARA-AZUMI, School of Engineering,
Tokyo Institute of Technology, Japan

Recently, the quest to reduce energy consumption in digital systems has been the subject of a number of ongoing studies. One of the most researched focuses is *approximate computing* (AC). AC is a new computing paradigm in both hardware and software designs that aim to achieve energy-efficient digital systems. Although a variety of AC techniques have been studied so far, the main question, “How (in which section) can a program or a circuit be approximated?”, has not been answered yet. This work addresses the above issue by developing a software framework Static Significance Analysis for Approximate Computing (SSA-AC) to analyze the target application program and guide the designers to identify parts of the program to which approximation can or cannot be applied. SSA-AC statically analyzes the significance of variables in the precise version of the program and thus needs no trial-and-error evaluation or specific test data. Experimental results show that SSA-AC can successfully extract the significance ranking of inputs/variables to be approximated in much shorter time than existing statistical works that are inevitably data dependent.

CCS Concepts: • **Computer systems organization** → **Embedded software**;

Additional Key Words and Phrases: Approximate computing, significance analysis, framework, path extraction

ACM Reference format:

Sara Ayman Metwalli and Yuko Hara-Azumi. 2019. SSA-AC: Static Significance Analysis for Approximate Computing. *ACM Trans. Des. Autom. Electron. Syst.* 24, 3, Article 34 (April 2019), 17 pages.
<https://doi.org/10.1145/3314575>

1 INTRODUCTION

The appearance of new technologies such as the Internet of Things (IoT), which aims to connect various types of devices into a common platform for them to communicate and exchange data freely, is leading to an explosion in the amount of data to be processed, computed, and stored. Hence, more computing energy and executing time will be needed to handle this explosion of data. Because of this, a variety of studies on new technologies have been made aiming to reduce the energy needed to compute/store a large volume of data (Bahga and Madiseti 2014).

This work is partially supported by JSPS KAKENHI 17H04677.

Authors' addresses: S. A. Metwalli and Y. Hara-Azumi; South Bldg. 3-317, 2-12-1 Ookayama, Meguro 152-8552, Tokyo, Japan; emails: {sara, hara}@cad.ict.e.titech.ac.jp.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Association for Computing Machinery.

1084-4309/2019/04-ART34 \$15.00

<https://doi.org/10.1145/3314575>

One of the most promising paradigms to solve the above problem is *approximate computing* (AC). AC presents that relaxing the application's accuracy at the hardware and/or software levels will save the energy consumed by these IoT applications. At the hardware level, approximate circuits such as approximate adders (Miao et al. 2012), multipliers (Rehman et al. 2016), and other logic circuits (Venkataramani et al. 2012) can improve hardware performance, leading to reduce energy consumption. For example, approximate adders may ignore the carry chain by allowing their sub-adders to perform in parallel. Similarly, approximate storage can store data approximately by truncating some lower-bits in floating-point data (Sampson et al. 2014) and exploiting unreliable memories like DRAM with less refresh rate and SRAM with lowered supply voltage (Raha et al. 2017). At both the hardware and software levels, algorithmic approximation such as memoization (Keramidas et al. 2015; Osawa and Hara-Azumi 2017) and perforation (Alippi 2014), where some iterations of loops can be skipped and similar data can be reused, will enable us to process the target application quickly. Also, computations can be reformulated to achieve accelerations on specialized hardware (e.g., a neural processing unit) (Han and Orshansky 2013; Venkataramani et al. 2015). A recent survey paper (Jiang et al. 2017) has summarized the latest AC techniques. Interested readers are referred to it for further details.

The concept of AC can provide significant contributions to a variety of application domains by reducing the computing time and energy needed to process the given application. When applying AC, however, one has to carefully consider the following three questions: (1) "Which parts of the application can be approximated without significantly affecting the quality of the application's results?," (2) "To what degree can these parts be approximated?," and (3) "What is the best approximation technique that can be applied to maximize the benefits of using AC?" So far, a number of existing works, including the ones mentioned above, have been addressing AC techniques regarding question (3) by showing how these techniques can reduce the execution time and energy at different levels of designs assuming that the approximatable parts and degree have been already identified. However, questions (1) and (2) have not been answered sufficiently yet.

In our work, we aim to answer the aforementioned remaining questions and analyze applications at the software level of the design. As the first step, this article focuses on the question (1) "*Which parts of the application can be approximated?*" and presents an analytic framework to determine which inputs/variables (i.e., parts of the application program) have a more significant effect (i.e., significance) on the quality of the application's output(s) than others. More specifically, our work first statically analyzes the application program and symbolically expresses the output(s) only with designer-specified inputs/variables. Then, those inputs/variables are comprehensively evaluated concerning operation types and appearance to generate the significance ranking. Evaluations demonstrate that our work can successfully determine the significance of inputs/variables within much shorter time—an average of 88% and up to 97% reduction—than existing statistical works. To the best of our knowledge, this is the first work to address the static analysis of applications toward AC focusing on the question (1) *with no need of time-consuming trial-and-error evaluation based on specific AC techniques and a great amount of test data* (as discussed in more details in the next section).

The contributions of our article are as follows:

- We propose a novel framework called SSA-AC that statically analyzes a software program toward approximate computing (AC). Our framework is based on the static path extraction and generates a significance ranking list of inputs/variables based on their effect on the output(s) of a given application.
- The significance analysis is done comprehensively considering how variables are used (i.e., operations type and coefficients) and how often these variables appear (i.e., occurrence and path probability).

- By statically analyzing a program through the path extraction, our framework can resolve two crucial issues residing in conventional statistical approaches: the heavy dependency on test data and time-consuming trial-and-error evaluations. Even though our framework does not rely on the test data or trial-and-error evaluations, it can obtain an equivalent significance ranking of inputs/variables to statistical approaches. Furthermore, our work is more scalable to the size of programs.
- It can be also seen that our work is orthogonal to conventional statistical works in that our work can efficiently prune the parts of a given program that cannot be candidates for approximation and thus do not need further analysis. In other words, through our analysis, the designers can efficiently perform statistical analysis only on necessary parts of the program and save the analysis time.
- We make our framework, SSA-AC, open source and accessible at (SSA-AC [n.d.](#)) to encourage and contribute to the community.

The remainder of this article is organized as follows. Section 2 briefly explains related works and the motivation of our work. Sections 3 and 4 describe the overview of our analysis framework and analysis techniques that are used in the framework. Section 5 demonstrates the effectiveness of our work over existing works. Finally, Section 6 concludes this article.

2 STATISTICAL WORKS AND MOTIVATION

Approximate computing techniques can be applied to different/multiple levels of the design layers, i.e., at both/either hardware (circuit and architecture) and/or software levels. To effectively utilize AC techniques, the program analysis before approximation is essential. Most existing works targeting both/either of hardware and software have addressed the analysis on the quality of the application's output(s) *by employing AC techniques*. Here we briefly point out some recent state-of-the-art works.

MACACO (Venkatesan et al. 2011) is a tool that analyzes how an approximate circuit behaves with reference to the precise (i.e., original) circuit. The goal of this work is to help the hardware designers evaluate the impact of approximating circuits at the hardware level. iACT (Mishra et al. 2014) is a framework developed by Intel to target multiple levels of the design. It is a compiler-based runtime framework that embeds the programmer-annotated AC knobs in the binary for approximate memoization and approximate circuits. High-level pragmas provide hints to the compiler and the adaptive runtime system to perform approximations on the desired functions within the desired quality. ASAC (Roy et al. 2014) performs a sensitivity analysis to automatically generate annotations for the program's variables toward AC. It extracts information about the sensitivity of the outputs for the given test data. ApproxIt (Zhang et al. 2014) is a framework to manage the quality of approximations at the system level by observing runtime errors. Since the effects of AC differ by different application domains, some other works target a specific application or application domain to study the quality of outputs. For example, the significance-driven AC for image processing is examined in Burke et al. (2017). AxBench (Yazdanbakhsh et al. 2017) provides various and broad representative benchmarks on different levels of the CPUs, GPUs, and hardware designs. It contains a variety of input datasets so that the designers can study and analyze the effects of different AC techniques. These benchmarks can be used in combinations with the aforementioned analysis works.

Existing works including the above ones (Mishra et al. 2014; Roy et al. 2014; Venkatesan et al. 2011; Zhang et al. 2014) try to figure out the best way of approximation using *statistical* approaches. These approaches follow trial-and-error evaluations at different levels—some operate at the hardware level, whereas the others operate at the software level. Such approaches considerably rely

Fig. 1. The overview of SSA-AC.

on specific test datasets. These datasets are either random, covering a wide range, but might be non-realistic or real data that depend on how the data sources are chosen. Either way, the strong dependency on the datasets largely affects analysis results. Also, because of a large number of trial-and-error runs, such statistical approaches are time-consuming. Therefore, it is difficult to expect satisfiable analysis on a large application program with a number of potential approximable variables (or parts of the program) within realistic time.

The motivation of this work is to present a novel analysis approach that resolves the issues of the data dependency and time consumption in these existing statistical works. Detailed explanations of our work will be given in the following sections.

3 SSA-AC FRAMEWORK

This section describes SSA-AC, the framework that we propose for analyzing the applicability of AC to a target application program. SSA-AC aims to resolve the following two crucial issues confronted in the conventional statistical analysis: *the time consumed in the trial-and-error evaluations* and *the heavy data-dependency on the given test data*. In this article, as the first step of our work, SSA-AC statically analyzes a given precise program, particularly focusing on the question (1) stated in Section 1, i.e., “which parts of the application can be approximated without significantly affecting the quality of the application’s results?”

3.1 Overview

The overall analysis flow that we propose (including SSA-AC) consists of three stages as shown in Figure 1: the pre-processing stage, the AC analysis (SSA-AC), and the post-processing stage. Here we briefly explain what is performed in each stage.

- **Pre-processing stage:** First, the pre-processing stage takes as input a precise (original) program (described as “Precise Program (INPUT)” in Figure 1). If necessary, then program slicing (i.e., a technique to trim parts of the program that are unnecessary for the following analysis) is conducted on the program to reduce the program complexity. Then, the designers manually apply the float-to-fixed point conversion, the definition of inputs and independent variables whose significance the designers would like to find on the output(s) and the parameters setting (e.g., loop iteration constraint) to the input program. In this work, we assume that variables (i.e., inputs and intermediate variables calculated by the inputs¹) are potential for approximation, and hence the designers are required to designate them to be symbolized as described in Listing 1. In the example code snippet, three variables “a,” “b,” and “c” are defined as symbols to express the output “z.” Similarly, a variable “d” is also defined as a symbol, because it is used in the conditional statements. We use

¹Intermediate variables should be symbolized and analyzed if the underlying hardware resources are intended to be approximated.

```

int main()
{
    //Define variables
    int a, b, c, d, w, y, z;
    //Make variables symbolic
    klee_make_symbolic(&a, sizeof(a), "a");
    klee_make_symbolic(&b, sizeof(b), "b");
    klee_make_symbolic(&c, sizeof(c), "c");
    klee_make_symbolic(&d, sizeof(d), "d");
    w = a/2;
    y = b/4;
    if (d == 0){
        z = 2*w + y;
        //The output in KQuary
        klee_print_expr("In case d=0, z=", z);
    }
    else if (d > 0){
        z = w + y + 6;
        //The output in KQuary
        klee_print_expr("In case d>0, z=", z);
    }
    else{
        z = 4*a - c + 6;
        //The output in KQuary
        klee_print_expr("In case d<0, z=", z);
    }
    return 0;
}

```

Listing 1. A example code snippet.

built-in functions of the underlying analysis engine to define the symbols and print out the outputs. Through applying these manual adjustments and settings, the “Precise Program (Analyzed)” in Figure 1 gets prepared and fed to the AC analysis stage.

- **AC analysis stage:** In the AC analysis stage, SSA-AC statically extracts path conditions and performs the significance analysis on the given program to generate a list of inputs/variables sorted in descending order of significance (i.e., *relatively* how much they contribute to the quality of the output(s)). More in-depth explanations on SSA-AC will be given in Section 4.
- **Post-processing stage:** Finally, according to the analysis results above, the post-processing stage (an optional stage) guides the designers to apply AC or the conventional statistical approach (e.g., Mishra et al. (2014), Roy et al. (2014), and Venkatesan et al. (2011)) to further analyze the program with specific AC techniques under designer-specified error metrics. SSA-AC can significantly reduce the exploration time of the post-processing stage that the designers or existing works have been suffering from. It should be noted that in spite of using less “inputs,” SSA-AC can provide listing results equivalent to existing works—SSA-AC needs only a precise program, whereas existing works need not only the precise program but also a large amount of test data and the AC-applied program (the details will be provided in Section 5).

As can be seen from Figure 1, SSA-AC contains two steps: “path extraction” and “significance analysis.” The explanations of the path extraction (i.e., the approach and the input/output descriptions) will be given in the remainder of this section, whereas those of the significance analysis will be left for Section 4.

3.2 Path Extraction

SSA-AC first statically extracts final output(s) of the program along with paths (i.e., combinations of conditional branches). They are essential information in the following significance analysis.

in case $d = 0$: $z =$: (Add w32 (ReadLSB w32 0 a) (SDiv w32 (ReadLSB w32 0 b) 4))	in case $d > 0$: $z =$: (Add w32 6 (Add w32 (SDiv w32 (ReadLSB w32 0 a) 2) (SDiv w32 (ReadLSB w32 0 b) 4)))	in case else: $z =$: (Add w32 6 (Sub w32 (Mul w32 2 (ReadLSB w32 0 a)) (ReadLSB w32 0 c)))
--	--	--

Fig. 2. Output representation in KQuery for the code snippet in Listing 1.

Because the final output(s) of the program may differ by the paths taken, the values of array elements, etc., SSA-AC utilizes part of an existing symbolic execution engine, KLEE (Cadaru et al. 2008), to extract such information.²

KLEE (Cadaru et al. 2008) is a symbolic virtual machine built on top of the LLVM compiler infrastructure (Lattner 2002). The core of KLEE is an interpreter loop that selects a state (a section of the input code) to run and then symbolically executes a single instruction in the context of the state. Thus, within KLEE, the results of executing the program are expressed in the form of symbols of inputs/variables and possible outcomes of conditional branches instead of actual values. This loop continues until no state remains or a designer-defined timeout is reached. Because KLEE is based on a constraint solver to define the relations between variables (Cadaru et al. 2015), these relations indicate constraints in a constraint solver language called KQuery, where each constraint represents a condition of an optimization problem that the solution must satisfy.

Although KLEE was originally developed for the purpose of testing/verifying a given program and generates test benches of the extracted paths, in this work, SSA-AC utilizes KLEE *only to make use of its static path extraction, not its test benches*. In other words, what we extract from KLEE is only *its symbolic representations on the outputs of the given program*. Out of a number of existing SEE tools such as in Xie et al. (2005) and Păsăreanu and Rungta (2010), we selected KLEE, because it is based on LLVM as well as a lot of other programming frameworks that have been developed recently—sharing a common platform and integrating technologies on it should be practical and useful from the designers' perspective.

3.3 Input and Output Descriptions of Path Extraction

As described in Figure 1, the path extraction takes as input the program with some manual adjustments and settings and outputs path extraction results. The extracted paths are expressed in the KLEE's constraint solver language, KQuery, which is a textual representation of constraint expressions. Figure 2 shows the output description for the code snippet in Listing 1. In KQuery, the output “z” is described in three different expressions that correspond to three exclusive conditions (or execution paths of the program), $d = 0$, $d > 0$, and else ($d < 0$). Each expression is composed of a string that states the path condition and the value of the output associated with the path. Table 1 summarizes representations in KQuery that appear in Figure 2 and those that frequently appear in general applications (Cadaru et al. 2008). For example, in the representation “(ReadLSB w32 0 a),” the included terms indicate are as follows: “ReadLSB” refers to a memory access, “w32” is the data size, “0” is the address of the variable in the virtual memory, and “a” is the name of the variable.

Here summarizes how a variety of coding cases and the designer-designated requirements should be handled in the input program and are then expressed in the output representation in KQuery.

²There may be alternative methods to express outputs of a program apart from actual values such as Abstract Syntax Tree (AST) (Wile 1997). AST represents the abstract syntax of a given program in the form of trees for the purpose of syntax analysis before compilation. In this work, we utilized KLEE to extract paths rather than AST, because KLEE can directly extract information that is necessary for the following significance analysis (e.g., paths taken and the values of array elements) in the form of symbols, whereas AST would need compensations to extract such information.

Table 1. Representations in the KQuery Language (See Cadar et al. (2008) for More Details)

KQuery representation	Definition
ReadLSB/ReadMSB	Memory access (read the least/most significant bits.) Depending on the underlying processor, either of them is used.
Add/Sub/Mul/SDiv	Operations (addition/subtraction/multiplication/signed division)
w32/w16/w8	Data size (32-bit/16-bit/8-bit)

- **Data Types:** Various types of variables like integer, short, and char are supported and expressed as w32, w16, and w8, respectively (see Table 1). Currently, only char and integer types of up to 32 bits are supported in KLEE (i.e., floating-point variables are not supported in the version 1.4.0). Therefore, to analyze floating-point variables, they need to be manually converted to fixed-point variables in advance in the pre-processing stage as mentioned in Section 3.1. In the later version of KLEE where floating-point variables are supported through an additional add-on, this manual conversion will become unnecessary.
- **Arrays:** In KQuery, an array can be also symbolized. For example, if we have an array of integers “a[N],” then it will be expressed in KQuery as “a[N]: w32.” When performing the significance analysis in Section 4, the array is treated as one variable (e.g., “a”).
- **Loops:** Most programs contain loop statements (e.g., for and while). KLEE handles the loops based on their situation. If the loop bound is infinite or more prominent than a “threshold,” N , then only the first N iterations are considered. Otherwise, all iterations are run. The designers have an option to control it by using a KLEE built-in function “klee_assume().” As mentioned in Section 3.1, this function has been directly specified in the input program to bound unbounded loops in bounded ones or to decrease the number of loop iterations. Note that setting a small threshold can reduce analysis time but *approximates* the analysis results. In other words, this threshold trades off the accuracy and analysis time. From careful examinations, in this work, we set the threshold to 200. Since realistic applications have bounded loops with larger iteration counts or unbounded loops, the threshold of 200 can well-balance these two effects, i.e., enabling us to analyze programs within a reasonable time while mitigating the accuracy degradation.
- **Dependent Variables:** In some cases, the designers consider intermediate computational results (or dependent variables expressed by independent inputs, e.g., variables “w” and “y” in Listing 1) as AC candidates and would like to find their significance. In KLEE, even if the designers symbolize such dependent variables by the function klee_make_symbolic(), they will be expressed with their precedent independent inputs irrespective of whether the built-in function symbolizes those inputs or not. In other words, eventually “a” and “b” are used in the expression of “z” instead of “w” and “y.” We satisfy the designers’ requirement as follows: The designers designate the output of not only “z” but also “w” and “y” by the function klee_print_expr(); KLEE then outputs the symbolic expressions of these three variables in KQuery; and, finally, “w” and “y” are integrated in the expression of “z” by replacing the corresponding expressions of “w” and “y.” This transformation can be used for the case that the designers assume AC-applied underlying hardware (Raha et al. 2017).
- **Pointers:** In many cases, the programs use pointers. Straightforward symbolization of such pointer variables only indicates their virtual memory location in KLEE. If the designers would like to examine the significance of variables pointed by pointers, then alias analysis needs to be performed on those pointers in advance.

Further details and concrete examples will be provided in the documents of our framework (SSA-AC n.d.).

4 SIGNIFICANCE ANALYSIS

Significance analysis (or “sensitivity analysis”) is a technique used to determine how different values of independent variables impact a particular resultant variable (Vassiliadis et al. 2016). It predicts the outcome of changes made on the variables by revealing how accurate each of the used variables needs to be and aids in the understanding of how the program behaves based on those variables. This is done through pinpointing sensitive variables that need extra care for changes, because even a small change in the sensitive variables can greatly affect the quality of the outputs.

In this work, we use this technique to evaluate the *relative sensitivity* among the used variables when outputs are allowed to have some error. In other words, we clarify variables whose changes cause less error than the others—such variables are candidates of approximation. We perform this analysis using the symbolic representation (explained in the previous section) so that we can flexibly handle condition changes that may occur to the program after the system deployment (e.g., acceptable error and inputs to the program). Unlike conventional statistical analysis, even if such changes occur, our approach does not have to re-perform the significance analysis.

There are generally two approaches to conduct significance analysis: inputwise (i.e., tracing the application’s paths from inputs to outputs) and outputwise (i.e., traversing the paths back from outputs to inputs). We selected the former to straightforwardly integrate the information of path conditions into the significance analysis as explained in the following subsections. By doing so, we can also successfully identify and narrow down the approximatable parts of the target application program.

Our significance analysis is hierarchically employed. First, it is applied to each execution path of the program, where three types of analysis are conducted: the linear regression coefficients (LRC) analysis, the occurrence-based analysis, and the operation-based analysis. Then, analysis results on all possible paths are integrated to comprehensively consider the significance of inputs/variables.

4.1 Path-based Analysis

Generally, a program has multiple paths to be executed by combinations of conditional branches. For each path p , our work first analyzes the significance of inputs/variables symbolized by the designers (see Figure 1) as follows.

4.1.1 Linear Regression Coefficients Analysis. Linear regression coefficients (LRC) (Seber and Lee 2012) are estimates of the significance of changes in the independent variables to a resultant variable and have been widely used in a variety of analytic methods (i.e., the LRC analysis). The “linearity” in its name refers to just the linear combination of the independent variables and the resultant variable. In other words, it can handle *not only linear operations but also nonlinear ones*,³ like trigonometric functions (e.g., $\sin()$) and exponential (Cohen et al. 2002; Ayyub and McCuen 2011).

We calculate the significance (hereafter, we interchangeably use “significance” and “weight” for the same meaning) of the variables through the LRC analysis. This is because different coefficients naturally make different errors even if the operation type of two operations is the same. For example, in the statement “ $z = 1024 \times x + 2 \times y$,” the error on the resultant variable “ z ” caused by approximating variables “ x ” and “ y ” is different, since “ x ” generally has the larger significance (i.e., more sensitivity) than “ y .”

The LRC analysis expresses the weight of a variable as a normalized value based on the coefficient of the variable against those of all variables. If the j th variable in the path p is “ x ,” the weight

³The independent variables can be linear or nonlinear, whereas only the resultant variables are restricted to be linear.

of “x,” $W_{LRC}(x, p)$, is calculated using Equation (1):

$$W_{LRC}(x, p) = \frac{B_j}{\sum_{j'}^J B_{j'}}, \quad (1)$$

where B_j represents the coefficient of the j th variable (“x”) from J independent variables. For example, the statement of “z” in the first path p_1 of Figure 1, the weights of variables “a” and “b” are $W_{LRC}(a, p_1) = 1/(1 + 1/4) = 4/5 = 0.8$ and $W_{LRC}(b, p_1) = 1/4/(1 + 1/4) = 1/5 = 0.2$, respectively.

This calculation shows the extent to which an independent variable explains the variation in the resultant variable in the data. Whereas the regression coefficient identifies differences in significance, the significance is also influenced by the field-use or environment of the application (i.e., the range of input variables). Assuming that the designers cannot predict such environments at the design time, this work handles abstract weights for general use. Our future work will let the designers to optionally designate the range of input variables if they are predictable at the design time.

4.1.2 Occurrence-based Analysis. In case a program has only one output, all inputs/variables are likely to have the same occurrence based on their appearances in the output⁴. However, when the program has multiple outputs and not all inputs/variables appear in all of them, the occurrence-based analysis is useful. For example, in two outputs “w = x + 2y” and “z = 4x” on path “p,” the occurrence of “x” and “y” is 2 and 1, respectively. Defining the occurrence-based weight of the variable “x” on path “p” as $W_{OCB}(x, p)$, the weights of variables “x” and “y” are calculated as $W_{OCB}(x, p) = 2/(2 + 1) = 0.67$ and $W_{OCB}(y, p) = 1/(2 + 1) = 0.33$, respectively. In other words, to calculate the occurrence-based weight, we sum up the number of appearances of individual variable in each of the program outputs and normalize them.

It should be noted that, in some cases, a variable would appear more than once in an expression, such as “z = 2x + 4y + x/y” where further simplification cannot be done. This case affects the LRC-based analysis, not the occurrence-based analysis. In this case, for the variable “x,” the LRC-based weight is $W_{LRC}(x, p) = 0.43(= (2 + 1)/(2 + 4 + 1))$, whereas the occurrence-based weight is $W_{OCB}(x, p) = 0.5(= 1/(1 + 1))$.

4.1.3 Operation-based Analysis. The last analysis is the operation-based analysis where we consider the type of operations used in the output expression. In a program, there are generally multiple types of operations that can be divided into two groups: arithmetic operations and logic operations. In each group, different operations have different levels of significance. For example, in arithmetic operations (addition, subtraction, multiplication, and division), the significance level of each operation is generally used to decide which operation is performed first (or prioritized) if more than one type of operations occurs in the same expression. Here, let us consider a calculation of “z = 3 + 2×5 − 3 + 6/2.” The levels of operations control how this calculation should be done. Since the priority in arithmetic operations is in the order of multiplication/division and addition/subtraction, the value of “z” in this example is 13.

In this work, the operation-based analysis is performed in a somewhat similar approach. To get the weight of an operation, we consider four levels of significance as defined in Table 2, where $a_1 \leq a_2 \leq a_3 \leq a_4$. What is different than the aforementioned operational priority is that we prioritize (i.e., give larger weights to) operations whose outputs will use larger bit-width. Thus, multiplication (a_4) is given a larger weight than division (a_2). Here we define the operation-based weight of the variable “x” on the path “p” as $W_{OPB}(x, p)$. If the weights are set to be $a_1 = 1$, $a_2 = 2$,

⁴We assume that statements like “z = 2x + 4x” are simplified and converted to “z = 6x” (i.e., the occurrence of “x” in this statement is one).

Table 2. Weights for Operation-based Analysis

Significance	Weight	Operation type
Highest	a_4	Multiplication, shift left
Medium high	a_3	Addition, subtraction
Medium low	a_2	Division, shift right
Lowest	a_1	Logic operations

Table 3. Weight Calculation for the Example Program in Figure 1

Path Variables	$d = 0$			$d > 0$			else ($d < 0$)		
	a	b	c	a	b	c	a	b	c
LRC (W_{LRC})	0.8	0.2	0	0.7	0.3	0	0.7	0	0.3
Occurrence (W_{OCB})	0.5	0.5	0	0.5	0.5	0	0.5	0	0.5
Operation (W_{OPB})	0.5	0.5	0	0.5	0.5	0	0.5	0	0.5
Path-based Total (W_{PB})	1.8	1.2	0	1.7	1.3	0	1.7	0	1.3
Overall Total (W_O)	a: 1.7, b: 0.8, c: 0.4								

$a_3 = 3$, and $a_4 = 4$, then the operation-based weights of the variables “a” and “b” on the first path “ p_1 ” in Figure 1 are $W_{OPB}(a, p_1) = W_{OPB}(b, p_1)$, because they are both the operands of an addition operation. Then, through normalization, $W_{OPB}(a, p_1) = W_{OPB}(b, p_1) = 0.5$ is set. Complex mathematical functions, such as trigonometric and power functions, are decomposed to four arithmetic operations so that the weight can be expressed with the weights in Table 2 (a_1 to a_4).

4.2 Total Analysis

Through the aforementioned three types of analyses, for the code in Figure 1, the weights of each input/variable are obtained for each path as described in the third to fifth rows in Table 3 (for the operation-based analysis, the parameters were set to $a_1 = 1$, $a_2 = 2$, $a_3 = 3$, and $a_4 = 4$). Based on these results, an overall weight is calculated, which is twofold as follows.

- (1) **Path-based Total Weight (W_{PB}):** The total weight of the variable “x” is integrated in each path p as expressed by Equation (2). The results are described in the sixth row of the table,

$$W_{PB}(x, p) = W_{LRC}(x, p) + W_{OCB}(x, p) + W_{OPB}(x, p). \quad (2)$$

- (2) **Overall Total Weight (W_O):** Finally, the weights of the variable “x” in all paths are integrated considering the possibility of the path p , P_p , as expressed by Equation (3). The results are described in the bottom row of the table where for brevity all P_p ’s are assumed to be equal (i.e., 0.33),

$$W_O(x) = \sum_p P_p \times W_{PB}(x, p). \quad (3)$$

The path possibility P_p can be determined by either of three following ways: the designer’s designation, the profiles of path probability (or statistics)⁵ obtained in advance (Chilimbi et al. 2009), or the range analysis on branch variables (Gort and Anderson 2013). In the latter two ways, the path probabilities of all paths can be fed to the significance analysis as shown in Figure 1.

⁵If the designers cannot predict P_p at the design time, then they may quantitatively find which paths are more likely taken by using specific datasets. We do not prohibit to rely on statistics for setting P_p .

For codes with multiple outputs, the total weight is first calculated for each output by the path-based total weight W_{PB} of the paths relevant to each output. Then, the total weight W_O is calculated to each output similarly as explained above.

Recalling that our goal is to extract *the significance ranking* of inputs/variables to outputs, the designers need to look at *the relative values of W_O (or magnitude relationship between inputs/variables)*, not the absolute values of W_O themselves. For example, for the code in Figure 1, we obtain the significance ranking of three variables, which is “a,” “b,” and “c” in the descending order. This means that the variable “c” is the least significant or the most tolerable to approximation among these three variables.

As aforementioned, the static path extraction-based significance analysis that we presented above is the very first attempt to resolve the issues in time-consuming statistical counterparts. As the first step of this work, we intended to make the analytic models as simple as possible by starting with three fundamental measures above (i.e., the coefficient-based, operation-based and occurrence-based weights) (Mitchell and Sevitsky 2007) so that they are easily applicable to practically large, complex applications. Integrating more measures to improve the accuracy may sacrifice the simplicity of the models, resulting in the degradation of the applicability—they are in a trade-off. We will keep studying the best tradeoff between the simplicity and accuracy of the models in our future work.

5 EXPERIMENTS

This section demonstrates the effectiveness of our proposed framework, SSA-AC, over the conventional statistical approach. We first explain the experimental setup and then provide experimental results.

5.1 Experimental Setup

To evaluate the effectiveness of SSA-AC, we compare SSA-AC and the conventional statistical approach by the following two metrics:

- **Scalability:** We evaluate the scalability of SSA-AC and the statistical approach in terms of *analysis time*, varying the program size.
- **Accuracy:** We evaluate the accuracy in terms of *the significance ranking results* by comparing the outputs of applying AC to inputs/variables against those of executing the original program in precise computing. Here, we assume *least significant bits (LSBs) truncation* (Lee et al. 2017) as an AC technique (from 32-bit int to 16-bit short for integer inputs/variables, from 64-bit double to 32-bit float for floating-point inputs/variables, and so on).

Although some statistical analysis works have been presented recently, such as in Mishra et al. (2014), Roy et al. (2014), Zhang et al. (2014), and Venkatesan et al. (2011), there are no open source frameworks available for comparisons. Therefore, for our evaluation, we implemented an in-house analysis framework to represent them, where we follow the general analysis flow using randomly generated test datasets. For the test data generation, we implemented an advanced technique to achieve a good tradeoff between the coverage of the test data (or accuracy) and analysis time, similarly as a technique called “hyperbox” in ASAC (Roy et al. 2014) (the interested reader is referred to Roy et al. (2014) for its detail). Regarding the number of test datasets (runs#) for the statistical analysis, it is randomized such that it can be higher than the number of paths in the code yet less than 200. Since much more runs will be necessary in realistic cases, these runs can be regarded as the minimum ones.

We utilized six realistic benchmarks with different characteristics in operations, paths, independent inputs/variables (which we symbolized), and outputs, as summarized in Table 4. These

Table 4. Characteristics of Benchmarks

Benchmark	Operations#				Symb. Vars		
	Mult&L-Shft (a_4)	Add&Sub (a_3)	Div&R-Shft (a_2)	Logic (a_1)	Paths#	(In#/Var#)	Output#
BWLPF	27	23	8	5	7	3/0	1
Farrow Re-sampler	7	14	2	9	13	2/2	1
FPRIF	5	10	8	8	10	1/2	2
Heuristics MUW	12	43	4	14	24	5*/0	3
IIRBSF	7	11	2	5	16	1/3	1
NRLPF	4	13	8	8	5	3/0	2

*This includes one non-approximatable input (i.e., the remaining four inputs are approximation targets).

benchmarks are often used in the digital signal processing field. For each benchmark, we show brief descriptions, including the source and representative data types, below:

- **BWLPF**: Passive low-pass filter (LPF), implemented based on a recursive finite impulse response (FIR) filter with a circular buffer (Embree and Danieli 1998). Representative data types are 32-bit integer and 32-bit floating-point numbers. $\sin()$ and $\cos()$ are used from the math library.
- **Farrow Re-sampler**: Reconstructing continuous-time signals from samples using an impulse response that is approximated by piecewise polynomials (Smith 2010). Representative data types are 32-bit integer and 64-bit floating-point numbers. $\text{pow}()$ and $\text{floor}()$ are used from the math library.
- **FPRIF**: FIR filter with 63 taps using fixed-point numbers (Embree and Danieli 1998). Representative data types are 16-bit/32-bit/64-bit integer.
- **Heuristics MUW**: A heuristic power-allocation by multi-user waterfilling (a communication problem when scheduling transmissions from a base station to several receivers with frequency varying channels) (Giovannidis et al. 2006). Representative data types are 32-bit integer and 32-bit/64-bit floating-point numbers. $\text{floor}()$ and $\text{ceil}()$ are used from the math library. In the following evaluation of applying AC techniques, we excluded one out of five inputs, because the variable indicates the number of “users,” which is obviously not-approximatable.
- **IIRBSF**: A series of infinite impulse response (IIR) filters, where filter coefficients sets are generated using a specific center frequency (Embree 1995). Representative data types are 16-bit/32-bit integer and 64-bit floating-point numbers. $\text{pow}()$, $\text{sqr}t()$, $\text{cos}()$, and $\text{tan}()$ are used from the math library.
- **NRLPF**: Noise generator implemented as a non-recursive LPF with an output of a linear function of inputs (Embree 1995). Representative data types are 32-bit integer and 32-bit floating-point numbers.

In our evaluation, for simplicity and without loss of generality, we assume that the paths have the same possibility to be taken. We targeted all of inputs and independent variables (e.g., coefficient variables) for symbolization,⁶ which are described as “In#” and “Var#,” respectively, in the seventh

⁶While hardware designs would assume not only inputs and independent variables but also intermediate variables for approximation due to approximated hardware resources, software designs would assume only the former for approximation. As our framework can be used for both software and hardware designs, we dealt with only their common interests (i.e., inputs and independent variables) for symbolization.

Table 5. Analysis Time of the Statistical Approach vs. SSA-AC

Benchmark	Statistical approach				SSA-AC		
	Time/run (s)	Runs # (=Data# × AC tech.#)	Total (s)		SEE (s)	Analy. (s)	Total (s)
BWLPF	0.359	924 (=77 × 12)	331.716		0.547	0.335	0.882
Farrow Re-sampler	0.436	1,524 (=127 × 12)	664.464		0.765	0.324	1.089
FPRIF	0.832	290 (=58 × 5)	241.280		1.003	0.945	1.945
Heuristics MUW	1.985	1,236 (=103 × 12)	2,453.460		2.576	1.134	3.710
IIRBSF	1.243	516 (=43 × 12)	641.388		1.442	1.036	2.478
NRLPF	0.598	1,104 (=138 × 8)	660.192		0.898	0.683	1.581

Fig. 3. Evaluations on the accuracy of the significance ranking.

column of Table 4. For operation-based analysis, the weights for operation types were set as $a_1 = 1$, $a_2 = 2$, $a_3 = 3$, and $a_4 = 4$ (see Table 2).

5.2 Experimental Results

We first compare the scalability of the existing statistical approach and SSA-AC in terms of analysis time. Table 5 describes the results for the statistical approach and SSA-AC, in the second to fourth columns and the fifth to seventh columns, respectively. Note that, for each benchmark, the second column “Time/run” for the statistical approach describes average time (in seconds) per run to execute the precise (original) program and an AC-applied program for a set of test data and compare their output results.⁷ The total analysis times to be compared are in the fourth and seventh columns—the former is the product of time/run and runs# (in the second and third columns), while the latter is the summation of time taken in SEE and significance analysis (in the fifth and sixth columns). They are both the total time to get the ranking results (which will be disclosed and discussed later in Figure 3 and Table 6).

Although each run is quicker in the statistical approach than in SSA-AC, overall the former has longer analysis time due to a large number of trial-and-error evaluations incurred by comparisons of the precise (original) program and the AC-applied program under numerous combinations on where to apply AC techniques. For the statistical approach, recall that the runs# are set to be rather small in our evaluation, it generally takes much longer time, resulting in poor scalability for realistic benchmarks. Contrary, in spite of longer time per run, the total time in SSA-AC is much shorter. This is because SSA-AC requires a *single* execution of the precise program only. In our evaluation, SSA-AC achieved an average time savings of 85% and a maximum of 98% compared

⁷Because the results of outputs can differ by different test data, both of the precise and AC-applied programs are needed to be executed and compared for each set of test data.

Table 6. Significance Ranking Results: The Statistical Approach vs. SSA-AC

Benchmark	Description of inputs/variables	Stat. analy. Ave. acc. (%)	SSA-AC Sig. ranking
BWLPF	Filter order	100	LSV
	input frequency	50.4	MLSV
	Cut-off frequency	39.9	MSV
Farrow Re-Sampler	Delay coefficient	99.9	LSV
	Index in input sequence	99.0	MLSV2
	Look-up table	95.0	MLSV1
	Coefficient matrix	90.0	MSV
FPRIF	Filter length	100	LSV
	Filter Coefficients	44.7	MLSV
	Input matrix	21.1	MSV
Hueristics MUF	Processing order	100	LSV
	Number of transmissions	64.1	MLSV2
	Number of resources	50.7	MLSV1
	Power per user	37.7	MSV
IIRBSF	Coefficient factors	100	LSV
	Sampling frequency scaling	99.8	MLSV2
	Scaling factors	88.0	MLSV1
	Input matrices	41.6	MSV
NRLPF	Zero budding in output	100	LSV
	Input frequency	75.2	MLSV
	Cut-off frequency	17.5	MSV

with the statistical approach. Although the time for SSA-AC (especially for the SEE part) increases by growing the number of possible paths in the programs, it can be reasonably managed by leveraging existing techniques in suppressing the path explosion issue as discussed in Section 3.2, and thus further analysis time savings can be expected. Here we have confirmed that SSA-AC is more scalable for realistic benchmarks than the conventional statistical approach.

Next, Figure 3 describes the accuracy of the analysis results provided by SSA-AC. For the same six benchmarks, we applied LSBs truncation as AC techniques to all independent inputs/variables. Since the benchmarks used have up to four independent inputs/variables, according to the ranking results of SSA-AC, the figure refers the inputs/variables as the least significant variable (LSV), two different medium-level significant variables (MLSV1 and MLSV2)⁸ and the most significant variable (MSV). As aforementioned, LSBs truncation we applied was the conversion from 32-bit int to 16-bit short and from 64-bit double to 32-bit float for integer and floating-point inputs/variables, respectively. In the figure, the x -axis shows these four inputs/variables in the descending order of significance for each benchmark. The y -axis describes the accuracy (in %) normalized by the precise computing result of each benchmark. For benchmarks with multiple outputs, the accuracy is regarded as the average effect of all outputs. As seen from the figure, applying AC to MSV leads to a large accuracy degradation in all benchmarks. However, even when the same AC technique is applied to the other variables, especially LSV, the degradation is smaller or almost negligible. In all benchmarks, the effects on the output(s) (i.e., degradation degree) are in the ascending order

⁸MLSV1 is more significant than MLSV2. For benchmarks with three inputs/variables for symbolization, MLSV1 and MLSV2 represent the same variable. MLSV2 is thus omitted in those benchmarks.

from left (LSV) to right (MSV). These results indicate that SSA-AC can successfully extract the significance ranking of independent inputs/variables to the output(s).

Finally, we compared the ranking obtained by SSA-AC against that by the statistical approach. In the statistical approach, we applied different approximations (LSBs truncation) to different variables and evaluated the average accuracy—because the output results of each benchmark can differ by combinations of test data used and AC techniques applied, several AC techniques need to be applied to get the ranking results. Note that the analysis was done without the knowledge of the significance. After obtaining the significance ranking, we examined the significance of the inputs/variables that were recognized by SSA-AC as LSV, MLSV2, MLSV1, and MSV in Figure 3. Table 6 gives a brief description of these inputs/variables in each benchmark and compares the results of the statistical approach and SSA-AC. In each benchmark, the inputs/variables are listed in the ascending order of significance results obtained by SEEA-AC (i.e., LSV, MLSV2, MLSV1, and MSV) from top to bottom. The number of approximation techniques applied (i.e., the types of truncation) to each benchmark has been disclosed in the third column of Table 5. From Table 6, we can clearly see that SSA-AC successfully achieves the same significance ranking as the statistical analysis, in spite of only a single execution of the program and much shorter analysis time, demonstrating the usefulness of SSA-AC.

Recall that the goal of this work is to extract the significance ranking of inputs/variables toward AC, with no need of time-consuming trial-and-error evaluations and specific test data, through the evaluations above, we confirmed that SSA-AC successfully achieves this goal and shows useful contributions to our community. The limitation in our current work is that SSA-AC does not express the accuracy gap (or the approximatable degree) of inputs/variables, which appears in the results of the statistical analysis in Table 6. Currently, as aforementioned, we assume that SSA-AC is used to prune inputs/variables that cannot be AC candidates (e.g., variables referred to as MSV are unlikely to be approximatable) prior to the time-consuming statistical analysis—the statistical analysis can then concentrate only on variables that would be AC candidates. The current SSA-AC is still the first step for AC analysis and hence will be extended to tackle with the accuracy gap indication. For example, we are currently considering to perform an additional analysis only on the variables with low significance to draw error graphs depending on the different approximation techniques.

6 CONCLUSIONS

Approximate computing is becoming attractive, as it presents a promising solution to relax the complexity as well as to save the execution time and energy consumption of digital systems. Although AC is becoming a breakthrough of current computing issues, extracting parts of the target application to which AC can be applied is a crucial issue. This article presented a systematic analysis framework, SSA-AC, based on the path extraction toward AC attempting to answer the above question. SSA-AC supports the designers to find AC-applicable inputs/variables by analyzing the target application code statically and then generating a significance ranking of those inputs/variables. Unlike conventional statistical approaches that require trail-and-error evaluations with a massive amount of test data and result in being heavily time-consuming, SSA-AC requires only a single run to analyze the target application code, enabling effective reduction of the analysis time. To the best of our knowledge, this work presents the first path extraction-based approach for the significance analysis toward AC.

Our quantitative evaluations on realistic benchmarks have shown that SSA-AC can analyze the program codes in much shorter time than the conventional statistical approaches—with an average time saving of 85% and a maximum of 98%—while successfully achieving the same significance ranking. In addition, from the results, we showed that we can efficiently prune the wasteful

trial-and-error evaluations by utilizing SSA-AC prior to the conventional statistical analysis when statistical analysis still needs to be performed. These results well-demonstrate the scalability and accuracy of our SSA-AC, leading to the practical usefulness over conventional works.

The subjects of our future work are mainly twofold: (1) extending models to consider more features of the target application while keeping the simplicity (i.e., applicability to large, complex applications) and (2) integrating the consideration on the approximatable degree to resolve the remaining question “to what degree” and to accomplish the analysis work toward AC.

Our framework, SSA-AC, presented in this article is open source and accessible at (SSA-AC [n.d.](#)).

REFERENCES

- Cesare Alippi. 2014. *Intelligence for Embedded Systems*. Springer, Berlin.
- Bilal M. Ayyub and Richard H. McCuen. 2011. *Probability, Statistics, and Reliability for Engineers and Scientists*. CRC Press, Boca Raton, FL.
- Arshdeep Bahga and Vijay Madisetti. 2014. *Internet of Things: A Hands-On Approach*. VPT.
- Dave Burke, Dainius Jenkus, Issa Qiqieh, Rishad Sha, Shidhartha Das, and Alex Yakovlev. 2017. Significance-driven adaptive approximate computing for energy-efficient image processing applications. In *Proceedings of International Conference on Hardware/Software Codesign and System Synthesis*. 1–2.
- Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of USENIX Conference on Operating Systems Design and Implementation*. 209–224.
- Christian Cadar, Vijay Ganesh, Raimondas Sasnauskas, and Koushik Sen. 2015. Symbolic execution and constraint solving. In *Dagstuhl Reports*.
- Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. 2009. HOLMES: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*. 34–44.
- Jacob Cohen, Patricia Cohen, Stephen G. West, and Leona S. Aiken. 2002. *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*. Routledge.
- Paul Embree and Damon Danieli. 1998. *C++ Algorithms for Digital Signal Processing*. Pearson Education.
- Paul M. Embree. 1995. *C Algorithms for Real-Time DSP*. Prentice Hall PTR.
- Anastasios Giovanidis, Thomas Haustein, Yosia Hadisusanto, Aydin Sezgin, and Donghee Kim. 2006. Multiuser scheduling using equal power in allocated subcarriers for OFDM uplink. In *Proceedings of the 40th Asilomar Conference on Signals, Systems and Computers*. 1820–1824.
- Marcel Gort and Jason H. Anderson. 2013. Range and bitmask analysis for hardware optimization in high-level synthesis. In *Proceedings of the Asia and South Pacific Design Automation Conference*. 773–779.
- Jie Han and Michael Orshansky. 2013. Approximate computing: An emerging paradigm for energy-efficient design. In *Proceedings of the European Test Symposium*. 1–6.
- Honglan Jiang, Cong Liu, Leibo Liu, Fabrizio Lombardi, and Jie Han. 2017. A review, classification, and comparative evaluation of approximate arithmetic circuits. *J. Emerg. Technol. Comput. Syst.* 13, 4 (2017), 60:1–60:34.
- Georgios Keramidas, Chrysa Kokkala, and Iakovos Stamoulis. 2015. Clumsy value cache: An approximate memoization technique for mobile GPU fragment shaders. In *Proceedings of the Workshop on Approximate Computing*. 1:1–1:32.
- Chris Arthur Lattner. 2002. *LLVM: An Infrastructure for Multi-Stage Optimization*. Ph.D. Dissertation. University of Illinois at Urbana–Champaign.
- Seogoo Lee, Lizy K. John, and Andreas Gerstlauer. 2017. High-level synthesis of approximate hardware under joint precision and voltage scaling. In *Proceedings of the Conference on Design, Automation & Test in Europe*. 187–192.
- Jin Miao, Ku He, Andreas Gerstlauer, and Michael Orshansky. 2012. Modeling and synthesis of quality-energy optimal approximate adders. In *Proceedings of the International Conference on Computer-Aided Design*. 728–735.
- Asit Mishra, Rajkishore Barik, and Somnath Paul. 2014. iACT: A software-hardware framework for understanding the scope of approximate computing. In *Proceedings of the Workshop on Approximate Computing across the System Stack*.
- Nick M. Mitchell and Gary Sevitsky. 2007. Inspecting the runtime behavior of a program while minimizing perturbation (Feb. 2007). US Patent 7,178,131.
- Hisashi Osawa and Yuko Hara-Azumi. 2017. Approximate data reuse-based processor: A case study on image compression. In *Proceedings of the Symposium on Embedded Systems for Real-Time Multimedia*. 32–40.
- Corina S. Păsăreanu and Neha Rungta. 2010. Symbolic PathFinder: Symbolic execution of java bytecode. In *Proceedings of the International Conference on Automated Software Engineering*. 179–180.
- Arnab Raha, Soubhagya Sutar, Hrishikesh Jayakumar, and Vijay Raghunathan. 2017. Quality configurable approximate DRAM. *IEEE Trans. Comput.* 66, 7 (2017), 1172–1187.

- Semeen Rehman, Walaa El-Harouni, Muhammad Shafique, Akash Kumar, and Jörg Henkel. 2016. Architectural-space exploration of approximate multipliers. In *Proceedings of the International Conference on Computer-Aided Design*. 1–8.
- Pooja Roy, Rajarshi Ray, Chundong Wang, and Weng Fai Wong. 2014. ASAC: Automatic sensitivity analysis for approximate computing. In *ACM SIGPLAN Notices*, Vol. 49. 95–104.
- Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. 2014. Approximate storage in solid-state memories. *ACM Trans. Comput. Syst.* 32, 3 (2014), 9.
- George A. F. Seber and Alan J. Lee. 2012. *Linear Regression Analysis*. Vol. 329. John Wiley & Sons, New York, NY.
- Julius O. Smith. 2010. Physical audio signal processing. *Linear Predictive* (2010).
- SSA-AC. [n. d.]. Retrieved from <http://www.cad.ict.e.titech.ac.jp/index.php/about-us/software-release>.
- Vassilis Vassiliadis, Jan Riehme, Jens Deussen, Konstantinos Parasyris, Christos D. Antonopoulos, Nikolaos Bellas, Spyros Lalis, and Uwe Naumann. 2016. Towards automatic significance analysis for approximate computing. In *Proceedings of the International Symposium on Code Generation and Optimization*. 182–193.
- Swagath Venkataramani, Srmat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2015. Approximate computing and the quest for computing efficiency. In *Proceedings of the Design Automation Conference*. 120.
- Swagath Venkataramani, Amit Sabne, Vivek Kozhikkottu, Kaushik Roy, and Anand Raghunathan. 2012. SALSA: Systematic logic synthesis of approximate circuits. In *Proceedings of the Design Automation Conference*. 796–801.
- Rangharajan Venkatesan, Amit Agarwal, Kaushik Roy, and Anand Raghunathan. 2011. MACACO: Modeling and analysis of circuits for approximate computing. In *Proceedings of the International Conference on Computer-Aided Design*. 667–673.
- David S. Wile. 1997. Abstract syntax from concrete syntax. In *Proceedings of the 19th International Conference on Software Engineering*. 472–480.
- Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. 2005. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 365–381.
- Amir Yazdanbakhsh, Divya Mahajan, Hadi Esmaeilzadeh, and Pejman Lotfi-Kamran. 2017. AxBench: A multiplatform benchmark suite for approximate computing. *IEEE Des. Test* 34, 2 (2017), 60–68.
- Qian Zhang, Feng Yuan, Rong Ye, and Qiang Xu. 2014. ApproxIt: An approximate computing framework for iterative methods. In *Proceedings of the Design Automation Conference*. 1–6.

Received July 2018; revised February 2019; accepted February 2019